

List of Challenging Experiments (Indicative)

1. Introduction Unix Commands	4 hours
2. Basic Shell Scripts	4 hours
3. Process Creation and execution	4 hours
4. CPU Scheduling Algorithms • FCFS, SJF, PRIORITY, Round Robin	4 hours
5. Write an algorithm to synchronize the agent and the smokers using semaphore.	4 hours
6. Producer–Consumer problem with Bounded Buffer	4 hours
7. Dining–Philosopher Problem	3 hours
8. Write an algorithm for synchronization between reader processes and write processes using semaphore.	3 hours

Ex.no:1 Introduction Unix Commands

Linux Directory Commands

1. pwd Command

The pwd command is used to display the location of the current working directory.

Syntax:

1. pwd

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ pwd
/home/javatpoint
```

2. mkdir Command

The mkdir command is used to create a new directory under any directory.

Syntax:

1. mkdir <directory name>

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ mkdir new_directory
javatpoint@javatpoint-Inspiron-3542:~$
```

3. rmdir Command

The rmdir command is used to delete a directory.

Syntax:

1. rmdir <directory name>

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ rmdir new_directory
javatpoint@javatpoint-Inspiron-3542:~$
```

4. ls Command

The ls command is used to display a list of content of a directory.

Syntax:

1. ls

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ ls
a          Desktop      examples.desktop  Music      sample
Akash      Directory    hello.c           pico       snap
a.out      Documents    hello.i           Pictures    Templates
composer.phar Downloads    hello.o           project    Test.txt
Demo.sh    eclipse      hello.s           Public     Videos
Demo.txt   eclipse-installer index.html        Python
Demo.txt~  eclipse-workspace mail             Python-3.8.0
```

5. cd Command

The cd command is used to change the current directory.

Syntax:

1. cd <directory name>

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ cd Desktop
javatpoint@javatpoint-Inspiron-3542:~/Desktop$
```

Linux File commands

6. touch Command

The touch command is used to create empty files. We can create multiple empty files by executing it once.

Syntax:

1. touch <file name>
2. touch <file1> <file2>

Output:

```
javatpoint@javatpoint-Inspiron-3542:~/Newfolder$ touch Demo.txt
javatpoint@javatpoint-Inspiron-3542:~/Newfolder$ touch Demo1.txt Demo2.txt
javatpoint@javatpoint-Inspiron-3542:~/Newfolder$ ls
Demo1.txt  Demo2.txt  Demo.txt
```

7. cat Command

The cat command is a multi-purpose utility in the Linux system. It can be used to create a file, display content of the file, copy the content of one file to another file, and more.

Syntax:

1. cat [OPTION]... [FILE]..

To create a file, execute it as follows:

1. cat > <file name>
2. // Enter file content

Press "**CTRL+ D**" keys to save the file. To display the content of the file, execute it as follows:

1. cat <file name>

Output:

```
javatpoint@javatpoint-Inspiron-3542:~/Newfolder$ cat > Demo.txt
This is a text file.
javatpoint@javatpoint-Inspiron-3542:~/Newfolder$ cat Demo.txt
This is a text file.
```

8. rm Command

The rm command is used to remove a file.

Syntax:

rm <file name>

Output:

```
javatpoint@javatpoint-Inspiron-3542:~/Newfolder$ rm Demo.txt
javatpoint@javatpoint-Inspiron-3542:~/Newfolder$ rm Demo1.txt Demo2.txt
```

9. cp Command

The cp command is used to copy a file or directory.

Syntax:

To copy in the same directory:

1. `cp <existing file name> <new file name>`

To copy in a different directory:

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ cp demo.txt demo1.txt
javatpoint@javatpoint-Inspiron-3542:~$ cp demo.txt Documents
```

10. mv Command

The mv command is used to move a file or a directory from one location to another location.

Syntax:

1. `mv <file name> <directory path>`

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ mv demo.txt Directory
```

11. rename Command

The rename command is used to rename files. It is useful for renaming a large group of files.

Syntax:

1. `rename 's/old-name/new-name/' files`

For example, to convert all the text files into pdf files, execute the below command:

1. `rename 's/\.txt$/\.pdf/' *.txt`

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ rename 's/\.txt$/\.pdf/' *.txt
javatpoint@javatpoint-Inspiron-3542:~$ ls
a                Desktop          examples.desktop  Music           Python-3.8.0
Akash            Directory        hello.c           Newfolder       sample
a.out           Documents        hello.i           pico            snap
composer.phar    Downloads        hello.o           Pictures         Templates
demo1.pdf        eclipse          hello.s           project         Test.pdf
Demo.sh          eclipse-installer index.html        Public          Videos
Demo.txt~       eclipse-workspace mail              Python
```

12. head Command

The head command is used to display the content of a file. It displays the first 10 lines of a file.

Syntax:

1. head <file name>

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ head Demo.txt
1
2
3
4
5
6
7
8
9
10
```

13. tail Command

The tail command is similar to the head command. The difference between both commands is that it displays the last ten lines of the file content. It is useful for reading the error message.

Syntax:

1. tail <file name>

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ tail Demo.txt
2
3
4
5
6
7
8
9
10
11
```

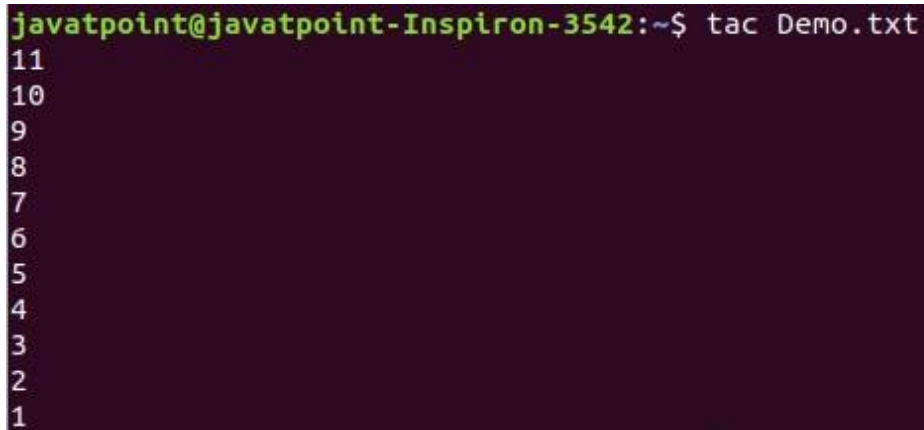
14. tac Command

The tac command is the reverse of cat command, as its name specified. It displays the file content in reverse order (from the last line).

Syntax:

1. tac <file name>

Output:



```
javatpoint@javatpoint-Inspiron-3542:~$ tac Demo.txt
11
10
9
8
7
6
5
4
3
2
1
```

15. more command

The more command is quite similar to the cat command, as it is used to display the file content in the same way that the cat command does. The only difference between both commands is that, in case of larger files, the more command displays screenful output at a time.

In more command, the following keys are used to scroll the page:

ENTER key: To scroll down page by line.

Space bar: To move to the next page.

b key: To move to the previous page.

/ key: To search the string.

Syntax:

1. more <file name>

Output:

```

;;; gyp.el - font-lock-mode support for gyp files.

;; Copyright (c) 2012 Google Inc. All rights reserved.
;; Use of this source code is governed by a BSD-style license that can be
;; found in the LICENSE file.

;; Put this somewhere in your load-path and
;; (require 'gyp)

(require 'python)
(require 'cl)

(when (string-match "python-mode.el" (symbol-file 'python-mode 'defun))
  (error (concat "python-mode must be loaded from python.el (bundled with "
    "recent emacsen), not from the older and less maintained "
    "python-mode.el")))

(defadvice python-indent-calculate-levels (after gyp-outdent-closing-parens
                                             activate)
  "De-indent closing parens, braces, and brackets in gyp-mode."
  (when (and (eq major-mode 'gyp-mode)
    (string-match "^ *[][]}][[],)]* *$"
      (buffer-substring-no-properties
--More--(7%)

```

16. less Command

The less command is similar to the more command. It also includes some extra features such as 'adjustment in width and height of the terminal.' Comparatively, the more command cuts the output in the width of the terminal.

Syntax:

1. `less <file name>`

Output:

```
;;; gyp.el - font-lock-mode support for gyp files.

;; Copyright (c) 2012 Google Inc. All rights reserved.
;; Use of this source code is governed by a BSD-style license that can be
;; found in the LICENSE file.

;; Put this somewhere in your load-path and
;; (require 'gyp)

(require 'python)
(require 'cl)

(when (string-match "python-mode.el" (symbol-file 'python-mode 'defun))
  (error (concat "python-mode must be loaded from python.el (bundled with "
    "recent emacsen), not from the older and less maintained "
    "python-mode.el")))

(defadvice python-indent-calculate-levels (after gyp-outdent-closing-parens
                                             activate)
```


Linux Filter Commands

17. cat Command

The cat command is also used as a filter. To filter a file, it is used inside pipes.

Syntax:

1. `cat <fileName> | cat or tac | cat or tac | . . .`

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ cat Demo.txt | tac | cat | cat | tac
1
2
3
4
5
6
7
8
9
10
11
```

18. cut Command

The cut command is used to select a specific column of a file. The '-d' option is used as a delimiter, and it can be a space (' '), a slash (/), a hyphen (-), or anything else. And, the '-f' option is used to specify a column number.

Syntax:

1. `cut -d(delimiter) -f(columnNumber) <fileName>`

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ cat >marks.txt
alex-50
alen-70
jon-75
carry-85
celena-90
justin-80
javatpoint@javatpoint-Inspiron-3542:~$ cut -d- -f2 marks.txt
50
70
75
85
90
80
javatpoint@javatpoint-Inspiron-3542:~$
```

19. grep Command

The grep is the most powerful and used filter in a Linux system. The 'grep' stands for "**global regular expression print**." It is useful for searching the content from a file. Generally, it is used with the pipe.

Syntax:

1. `command | grep <searchWord>`

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ cat marks.txt | grep 9  
celena-90
```

20. comm Command

The 'comm' command is used to compare two files or streams. By default, it displays three columns, first displays non-matching items of the first file, second indicates the non-matching item of the second file, and the third column displays the matching items of both files.

Syntax:

1. `comm <file1> <file2>`

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ comm Demo.txt Demo1.txt  
      1  
2  
      3  
comm: file 2 is not in sorted order  
    11  
      4  
      5  
    22  
    33  
6  
7  
8  
9  
comm: file 1 is not in sorted order  
10  
11
```

21. sed command

The sed command is also known as **stream editor**. It is used to edit files using a regular expression. It does not permanently edit files; instead, the edited content remains only on display. It does not affect the actual file.

Syntax:

1. `command | sed 's/<oldWord>/<newWord>/'`

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ echo class7 | sed 's/class/jtp/'  
jtp7  
javatpoint@javatpoint-Inspiron-3542:~$ echo class7 | sed 's/7/10/'  
class10
```

22. tee command

The tee command is quite similar to the cat command. The only difference between both filters is that it puts standard input on standard output and also write them into a file.

Syntax:

1. `cat <fileName> | tee <newFile> | cat or tac |.....`

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ cat marks.txt | tee new.txt | cat
alex-50
alen-70
jon-75
carry-85
celena-90
justin-80
javatpoint@javatpoint-Inspiron-3542:~$ cat new.txt
alex-50
alen-70
jon-75
carry-85
celena-90
justin-80
```

23. tr Command

The tr command is used to translate the file content like from lower case to upper case.

Syntax:

1. `command | tr <'old'> <'new'>`

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ cat marks.txt | tr 'prcu' 'PRCU'
alex-50
alen-70
jon-75
CaRRy-85
Celena-90
jUstin-80
```

24. uniq Command

The uniq command is used to form a sorted list in which every word will occur only once.

Syntax:

1. `command <fileName> | uniq`

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ sort marks.txt | uniq
alen-70
alex-50
carry-85
celena-90
jon-75
justin-80
```

25. wc Command

The wc command is used to count the lines, words, and characters in a file.

Syntax:

1. wc <file name>

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ wc marks.txt
 6  6 52 marks.txt
```

26. od Command

The od command is used to display the content of a file in different s, such as hexadecimal, octal, and ASCII characters.

Syntax:

1. od -b <fileName> // Octal format
2. od -t x1 <fileName> // Hexa decimal format
3. od -c <fileName> // ASCII character format

Output:

```

javatpoint@javatpoint-Inspiron-3542:~$ od -b marks.txt
00000000 141 154 145 170 055 065 060 012 141 154 145 156 055 067 060 012
00000020 152 157 156 055 067 065 012 143 141 162 162 171 055 070 065 012
00000040 143 145 154 145 156 141 055 071 060 012 152 165 163 164 151 156
00000060 055 070 060 012
00000064
javatpoint@javatpoint-Inspiron-3542:~$ od -t x1 marks.txt
00000000 61 6c 65 78 2d 35 30 0a 61 6c 65 6e 2d 37 30 0a
00000020 6a 6f 6e 2d 37 35 0a 63 61 72 72 79 2d 38 35 0a
00000040 63 65 6c 65 6e 61 2d 39 30 0a 6a 75 73 74 69 6e
00000060 2d 38 30 0a
00000064
javatpoint@javatpoint-Inspiron-3542:~$ od -c marks.txt
00000000 a l e x - 5 0 \n a l e n - 7 0 \n
00000020 j o n - 7 5 \n c a r r y - 8 5 \n
00000040 c e l e n a - 9 0 \n j u s t i n
00000060 - 8 0 \n
00000064

```

27. sort Command

The sort command is used to sort files in alphabetical order.

Syntax:

1. sort <file name>

Output:

```

javatpoint@javatpoint-Inspiron-3542:~$ sort marks.txt
alen-70
alex-50
carry-85
celena-90
jon-75
justin-80

```

28. gzip Command

The gzip command is used to truncate the file size. It is a compressing tool. It replaces the original file by the compressed file having '.gz' extension.

Syntax:

1. gzip <file1> <file2> <file3>...

Output:

```

javatpoint@javatpoint-Inspiron-3542:~$ gzip Demo.txt Demo1.txt
javatpoint@javatpoint-Inspiron-3542:~$ ls
a          Demo.txt.gz      examples.desktop  Music      Python-3.8.0
Akash      Desktop          hello.c          Newfolder  sample
a.out      Directory        hello.i          new.txt    snap
composer.phar Documents        hello.o          pico       Templates
demo1.pdf  Downloads        hello.s          Pictures    Test.pdf
Demo1.txt.gz eclipse          index.html       project     Videos
Demo.sh    eclipse-installer mail            Public
Demo.txt~  eclipse-workspace marks.txt       Python

```

29. gunzip Command

The gunzip command is used to decompress a file. It is a reverse operation of gzip command.

Syntax:

1. gunzip <file1> <file2> <file3>..

Output:

```

javatpoint@javatpoint-Inspiron-3542:~$ gunzip Demo.txt Demo1.txt
javatpoint@javatpoint-Inspiron-3542:~$ ls
a          Demo.txt~      examples.desktop  Music      Python-3.8.0
Akash      Desktop        hello.c          Newfolder  sample
a.out      Directory      hello.i          new.txt    snap
composer.phar Documents      hello.o          pico       Templates
demo1.pdf  Downloads      hello.s          Pictures    Test.pdf
Demo1.txt  eclipse        index.html       project     Videos
Demo.sh    eclipse-installer mail            Public
Demo.txt   eclipse-workspace marks.txt       Python

```

Linux Utility Commands

30. find Command

The find command is used to find a particular file within a directory. It also supports various options to find a file such as byname, by type, by date, and more.

The following symbols are used after the find command:

(.) : For current directory name

(/) : For root

Syntax:

1. find . -name "*.pdf"

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ find . -name "*.pdf"
./Test.pdf
./Python-3.8.0/Doc/library/turtle-star.pdf
./Akash/Joomla/Original Copy/Brochure-Joomla-2019.pdf
./Akash/Joomla/Original Copy/Joomla-Guide-Final.pdf
./local/share/Trash/files/2400966-250544e72f817db3bcef-1587140240830.pdf
./local/share/Trash/files/2400966-3ad982eaa58c5d43fb53-1585763620407.pdf
find: './.anydesk/incoming': Permission denied
./Downloads/ConfirmationPage_20030070774.pdf
./demo1.pdf
find: './.dbus': Permission denied
find: './.cache/dconf': Permission denied
./Directory/demo.pdf
./Directory/demo2.pdf
./Directory/demo1.pdf
```

31. locate Command

The locate command is used to search a file by file name. It is quite similar to find command; the difference is that it is a background process. It searches the file in the database, whereas the find command searches in the file system. It is faster than the find command. To find the file with the locates command, keep your database updated.

Syntax:

1. locate <file name>

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ locate sysctl.conf
/etc/sysctl.conf
/etc/sysctl.d/99-sysctl.conf
/etc/ufw/sysctl.conf
/snap/core/8935/etc/sysctl.conf
/snap/core/8935/etc/sysctl.d/99-sysctl.conf
/snap/core/9066/etc/sysctl.conf
/snap/core/9066/etc/sysctl.d/99-sysctl.conf
/snap/core18/1705/etc/sysctl.d/99-sysctl.conf
/snap/core18/1754/etc/sysctl.d/99-sysctl.conf
/usr/share/doc/procps/examples/sysctl.conf
/usr/share/man/man5/sysctl.conf.5.gz
```

32. date Command

The date command is used to display date, time, time zone, and more.

Syntax:

1. date

Output:


```
javatpoint@javatpoint-Inspiron-3542:~$ date
Fri May 22 21:51:05 IST 2020
```

33. cal Command

The cal command is used to display the current month's calendar with the current date highlighted.

Syntax:

1. cal<

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ cal
      May 2020
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

34. sleep Command

The sleep command is used to hold the terminal by the specified amount of time. By default, it takes time in seconds.

Syntax:

1. sleep <time>

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ sleep 4
```

35. time Command

The time command is used to display the time to execute a command.

Syntax:

1. time

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ time
real    0m0.000s
user    0m0.000s
sys     0m0.000s
```

36. zcat Command

The zcat command is used to display the compressed files.

Syntax:

1. zcat <file name>

Output:

```
javatpoint@javatpoint-Inspiron-3542:~$ ls
a                Demo.txt.gz      examples.desktop  Music           Python-3.8.0
Akash            Desktop          hello.c           Newfolder      sample
a.out            Directory        hello.i           new.txt        snap
composer.phar    Documents        hello.o           pico           Templates
demo1.pdf         Downloads        hello.s           Pictures        Test.pdf
Demo1.txt         eclipse          index.html        project        Videos
Demo.sh           eclipse-installer mail              Public
Demo.txt~         eclipse-workspace marks.txt         Python
javatpoint@javatpoint-Inspiron-3542:~$ zcat Demo.txt
1
2
3
4
5
6
```

37. df Command

The df command is used to display the disk space used in the file system. It displays the output as in the number of used blocks, available blocks, and the mounted directory.

Syntax:

1. df

Output:

```

javatpoint@javatpoint-Inspiron-3542:~$ df
Filesystem      1K-blocks    Used Available Use% Mounted on
udev             1931652         0   1931652  0% /dev
tmpfs            393260         1756   391504   1% /run
/dev/sda1       479668904 26471148 428762148  6% /
tmpfs           1966284     243536   1722748  13% /dev/shm
tmpfs            5120          4      5116   1% /run/lock
tmpfs           1966284         0   1966284   0% /sys/fs/cgroup
/dev/loop1       231936     231936         0 100% /snap/wine-platform-runtime/136
/dev/loop2       144128     144128         0 100% /snap/gnome-3-26-1604/98
/dev/loop4         384         384         0 100% /snap/gnome-characters/539
/dev/loop6       220160     220160         0 100% /snap/wine-platform-5-stable/4
/dev/loop5       164096     164096         0 100% /snap/gnome-3-28-1804/116

```

38. mount Command

The mount command is used to connect an external device file system to the system's file system.

Syntax:

1. `mount -t type <device> <directory>`

Output:

```

javatpoint@javatpoint-Inspiron-3542:~$ mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
udev on /dev type devtmpfs (rw,nosuid,relatime,size=1931652k,nr_inodes=482913,mode=755)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000)
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size=393260k,mode=755)
/dev/sda1 on / type ext4 (rw,relatime,errors=remount-ro)
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)

```

39. exit Command

Linux exit command is used to exit from the current shell. It takes a parameter as a number and exits the shell with a return of status number.

Syntax:

1. `exit`

Output:

```

javatpoint@javatpoint-Inspiron-3542:~$ exit

```

After pressing the ENTER key, it will exit the terminal.

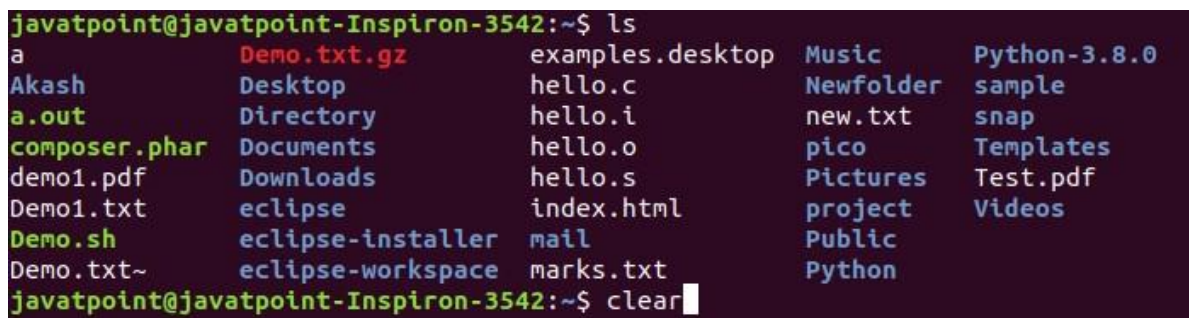
40. clear Command

Linux **clear** command is used to clear the terminal screen.

Syntax:

1. clear

Output:



```
javatpoint@javatpoint-Inspiron-3542:~$ ls
a                Demo.txt.gz      examples.desktop  Music           Python-3.8.0
Akash            Desktop          hello.c           Newfolder       sample
a.out            Directory        hello.i           new.txt         snap
composer.phar    Documents        hello.o           pico            Templates
demo1.pdf         Downloads        hello.s           Pictures         Test.pdf
Demo1.txt         eclipse          index.html        project         Videos
Demo.sh           eclipse-installer mail              Public
Demo.txt~         eclipse-workspace marks.txt          Python
javatpoint@javatpoint-Inspiron-3542:~$ clear
```

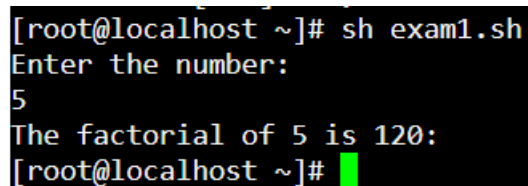
After pressing the ENTER key, it will clear the terminal screen.

Ex No 2: Basic Shell Scripts

2a) Write a Shell program to Calculate the Factorial of a Given Number

```
echo "Enter the number:"
read number
i=1
fact=1
while [ $i -le $number ]
do
fact=`expr $fact \* $i`
i=`expr $i + 1`
done
echo "The factorial of $number is $fact:"
```

OUTPUT:



```
[root@localhost ~]# sh exam1.sh
Enter the number:
5
The factorial of 5 is 120:
[root@localhost ~]#
```

2 b) Write a Shell Program for Interchanging Two Numbers

```
echo "Enter any two numbers"
read a
read b
echo "Before swapping $a $b"
c=$a
a=$b
b=$c
echo "After swapping $a $b"
```

OUTPUT:

```
Enter any two numbers
10
30
Before swapping      10      30
After swapping       30      10
```

2 c) Write a shell program to find the number is Armstrong Number or not.

```
echo "Enter the number"
read n
x=$n
sum=0
while [ $n -gt 0 ]
do
y=`expr $n % 10`
z=`expr $y \* $y \* $y`
sum=`expr $sum + $z`
n=`expr $n / 10`
done
if [ $x -eq $sum ]
then
echo " $x is an Armstrong number"
else
echo " $x is not an Armstrong number"
fi
```

OUTPUT:

```
Enter the number      153
153 is an Armstrong number
```

```
Enter the number      221
221 is not an Armstrong number
```

2 d) Write a shell program to find the Power value of a number

```
echo " Enter the value of x and n "
```

```

read x n
sum=1
i=1
while [ $i -le $n ]
do
sum=`expr $sum \* $x`
i=`expr $i + 1`
done
echo "The value of $x power $n is $sum"

```

OUTPUT:

```

Enter the value of x and n    3      2
The value of 3 power 2 is    9

```

2 e) Write a Shell program to find a number is Even or Odd.

```

echo "Enter the number "
read a
b=`expr $a % 2`
if [ $b -eq 0 ]
then
echo "$a is even"
else
echo "$a is odd"
fi

```

OUTPUT:

```

Enter the number 6
6 is even

```

```

Enter the number 7
7 is odd

```

2f) Write a Shell program to calculate the Sum and Average of a number.

```

echo "Enter the values "
i=1
sum=0
while [ $i -le 10 ]
do
echo "Enter number $i:"
read n
sum=`expr $sum + $n`
i=`expr $i + 1`
done
avg=`expr $sum / 10`
echo "The sum is $sum"
echo "The average is $avg"

```

OUTPUT

Enter the values:

```
Enter number 1:    3
Enter number 2:    5
Enter number 3:    2
Enter number 4:   76
Enter number 5:   45
Enter number 6:  325
Enter number 7:   78
Enter number 8: 2346
Enter number 9:   90
Enter number 10:  12
```

The sum is 2982

The average is 298

2g) Write a Shell program to find the Divisors of number.

```
echo "Enter a value"
read n
echo "The divisors of $n is"
i=1
t=`expr $n / 2`
while [ $i -le $t ]
do
temp=`expr $n % $i`
if [ $temp -eq 0 ]
then
echo $i
fi
i=`expr $i + 1`
done
```

OUTPUT

Enter a value 81

The divisors of 81 is

```
81
1
3
9
27
```

2h) Write a Shell program to find a string is Palindrome or not

```
echo "Enter the string"
read str
```

```

len=`echo $str | wc -c`
len=`expr $len - 1`
l=`expr $len / 2`
ctr=1
flag=0
while test $ctr -le $l
do
a=`echo $str | cut -c $ctr`
b=`echo $str | cut -c $len`
if test $a != $b
then
flag=1
break
fi
ctr=`expr $ctr + 1`
len=`expr $len - 1`
done
if test $flag -eq 0
then
echo "String is palindrome"
else
echo "String is not palindrome"
fi

```

OUTPUT:

Enter the string MALAYALAM
String is palindrome

Enter the string APPLE
String is not palindrome

2i) To write a shell program to find the Greatest of 3 numbers

```

echo "Enter the first number:"
read a
echo "Enter the second number:"
read b
echo "Enter the third number:"
read c
if [$a -gt $b -a $a -gt $c]
then
echo "$a is greater."
elif [$b -gt $c]
then
echo "$b is greater."
else
echo "$c is greater."
fi

```


OUTPUT:

```
Enter the first number:      15
Enter the second number:    14
Enter the third number:     16
```

16 is greater.

2 j) To write a shell program to generate the Fibonacci series for the given limit.

```
echo "Program to generate Fibonacci Series"
echo "Enter the range to be displayed"
read n
a=0
b=1
echo "Fibo Series"
echo $a
echo $b
i=2
while [ $i -lt $n ]
do
c=`expr $a + $b`
echo $c
i=`expr $i + 1`
a=$b
b=$c
done
```

OUTPUT

```
Enter the range to be displayed
6
Fibo Series
0
1
1
2
3
5
```

2 k) Write a shell program to find the Area and Circumference of Circle.

```
echo enter the radius
read r
area=`expr 22 / 7 \* $r \* $r`
circumference=`expr 2 \* 22 / 7 \* $r`
echo area= $area
echo circumference= $circumference
```

OUTPUT

```
[root@localhost ~]# sh file.sh
enter the radius
3
area= 27
circumference= 18
[root@localhost ~]#
```

Ex No :3. Process Creation and execution

Process creation is achieved through the **fork() system call**. The newly created process is called the child process and the process that initiated it (or the process when execution is started) is called the parent process. After the fork() system call, now we have two processes - parent and child processes.

Fork()

Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

Creates the child process. After this call, there are two processes, the existing one is called the parent process and the newly created one is called the child process.

The fork() system call returns either of the three values –

- Negative value to indicate an error, i.e., unsuccessful in creating the child process.
- Returns a zero for child process.
- Returns a positive value for the parent process. This value is the process ID of the newly created child process.

A) ORPHAN PROCESS:

PROGRAM:

```
#include<stdio.h>

main()
{
if(fork()==0)
{
system("clear");
printf("CHILD:I am child & my ID is : %d \n",getpid());
printf("CHILD: My parent is : %d \n",getppid());
}
else
{
printf("PARENT:I am parent: %d \n",getpid());
printf("PARENT: My parent is : %d \n",getppid());
}
}
```

OUTPUT:

CHILD: I am child & my ID is : 1959

CHILD: My parent is : 1

B) PARENT-CHILD PROCESS:

PROGRAM:

```
#include<stdio.h>
```

```

main()
{
if(fork()==0)
{
system("clear");
printf("CHILD:I am child & my ID is : %d \n",getpid());
printf("CHILD: My parent is : %d \n",getppid());
}
else
{
printf("PARENT: I am parent: %d \n",getpid());
sleep(2);
printf("PARENT: My parent is : %d \n",getppid());
}
}

```

OUTPUT:

```

PARENT: I am parent: 3012
CHILD: I am child & my ID is : 3013
CHILD: My parent is : 3012
PARENT: My parent is : 456

```

C)WAITING PROCESS:

PROGRAM:

```

#include<stdio.h>

main()
{
int pid,dip,cpid;
pid=fork();

```

```
if(pid==0)
{
printf("1st Child Process ID is %d\n",getpid());
printf("1st Child Process TERMINATING FROM THE MEMORY\n");
}
else
{
dip=fork();
if(dip==0)
{
printf("2nd Child Process ID is %d\n",getpid());
printf("2nd Child Process TERMINATING FROM THE MEMORY\n");
}
else
{
cpid=wait(0);
printf("Child with pid: %d is dead\n",cpid);
cpid=wait(0);
printf("Child with pid: %d is dead\n",cpid);
}
}
}
```

OUTPUT:

1st Child Process ID is 2562

1st Child Process TERMINATING FROM THE MEMORY

2nd Child Process ID is 2563

2nd Child Process TERMINATING FROM THE MEMORY

Child with pid: 2562 is dead

Child with pid: 2563 is dead

D)ZOMBIE PROCESS:

PROGRAM:

```
#include<stdio.h>

main()
{
int pid=fork();
system("clear");
printf("My ID is %d\n",getpid());
if(pid>0)
{
printf("My parent ID is %d\n",getpid());
sleep(05);
}
}
```

OUTPUT:

My ID is 2716

My parent ID is 2716

My ID is 2717

E) PARENT-CHILD PROCESS**PROGRAM:**

```
#include <stdio.h>

#include<sys/types.h>

#include<unistd.h>

int main()
{
    pid_t childpid;
    printf("\nPID: %d",getpid());
    printf("\nPARENT: %d",getppid());
    switch(childpid=fork())
    {
        case(pid_t)-1:perror("fork");
            break;

        case(pid_t)0:printf("\nCHILD CREATED\n");
            printf("CHILD PID: %d",getpid());
            printf("\nPARENT PID: %d",getppid());
            break;

        default: printf("\nPARENT AFTER FORK");
            printf("\nPARENT: %d",getpid());
```

```
        printf("\nCHILD: %d",childpid);  
    }  
    return 0;  
}
```

OUTPUT:

PID: 3264

PARENT:456

PARENT AFTER FORK

PARENT: 3624

CHILD: 3625 PARENT: 456

CHILD CREATED

CHILD PID: 3625

PARENT PID: 1

Ex No :4. CPU Scheduling Algorithms

4 a) Write an FCFS scheduling program in C to determine the average waiting time and average turnaround time given n processes and their burst times.

Program

```
#include <stdio.h>
int main()
{
    int pid[15];
    int bt[15];
    int n;
    printf("Enter the number of processes: ");
    scanf("%d",&n);

    printf("Enter process id of all the processes: ");
    for(int i=0;i<n;i++)
    {
        scanf("%d",&pid[i]);
    }

    printf("Enter burst time of all the processes: ");
    for(int i=0;i<n;i++)
    {
        scanf("%d",&bt[i]);
    }

    int i, wt[n];
    wt[0]=0;

    //for calculating waiting time of each process
    for(i=1; i<n; i++)
    {
        wt[i]= bt[i-1]+ wt[i-1];
    }

    printf("Process ID    Burst Time    Waiting Time    TurnAround Time\n");
    float twt=0.0;
    float tat= 0.0;
    for(i=0; i<n; i++)
    {
        printf("%d\t\t", pid[i]);
        printf("%d\t\t", bt[i]);
        printf("%d\t\t", wt[i]);
```

```

        //calculating and printing turnaround time of each process
        printf("%d\t\t", bt[i]+wt[i]);
        printf("\n");

        //for calculating total waiting time
        twt += wt[i];

        //for calculating total turnaround time
        tat += (wt[i]+bt[i]);
    }
    float att,awt;

    //for calculating average waiting time
    awt = twt/n;

    //for calculating average turnaround time
    att = tat/n;
    printf("Avg. waiting time= %f\n",awt);
    printf("Avg. turnaround time= %f",att);
}

```

Output:

```

[root@localhost ~]# gcc fcfs1.c
[root@localhost ~]# ./a.out
Enter the number of processes: 3
Enter process id of all the processes: 1
2
3
Enter burst time of all the processes: 2
4
1

```

Process ID	Burst Time	Waiting Time	TurnAround Time
1	2	0	2
2	4	2	6
3	1	6	7

```

Avg. waiting time= 2.666667
Avg. turnaround time= 5.000000[root@localhost ~]#

```

4 b) Write an SJF scheduling program in C to determine the average waiting time and average turnaround time given n processes and their burst times.

Program

```

#include<stdio.h>
int main()
{
    int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,totalT=0,pos,temp;

```

```

float avg_wt,avg_tat;
printf("Enter number of process:");
scanf("%d",&n);

printf("\nEnter Burst Time:\n");
for(i=0;i<n;i++)
{
    printf("p%d:",i+1);
    scanf("%d",&bt[i]);
    p[i]=i+1;
}

//sorting of burst times
for(i=0;i<n;i++)
{
    pos=i;
    for(j=i+1;j<n;j++)
    {
        if(bt[j]<bt[pos])
            pos=j;
    }

    temp=bt[i];
    bt[i]=bt[pos];
    bt[pos]=temp;

    temp=p[i];
    p[i]=p[pos];
    p[pos]=temp;
}

wt[0]=0;

//finding the waiting time of all the processes
for(i=1;i<n;i++)
{
    wt[i]=0;
    for(j=0;j<i;j++)
        //individual WT by adding BT of all previous completed processes
        wt[i]+=bt[j];

    //total waiting time
    total+=wt[i];
}

//average waiting time
avg_wt=(float)total/n;

printf("\nProcess\tBurst Time \tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)

```

```

{
    //turnaround time of individual processes
    tat[i]=bt[i]+wt[i];

    //total turnaround time
    totalT+=tat[i];
    printf("\np%d\t\t %d\t\t %d\t\t %d",p[i],bt[i],wt[i],tat[i]);
}

//average turnaround time
avg_tat=(float)totalT/n;
printf("\n\nAverage Waiting Time=%f",avg_wt);
printf("\n\nAverage Turnaround Time=%f",avg_tat);
}

```

Output

```

[root@localhost ~]# cc sjf.c
[root@localhost ~]# ./a.out
Enter number of process:3

Enter Burst Time:
p1:2
p2:4
p3:1

Process  Burst Time      Waiting Time      Turnaround Time
p3         1             0                 1
p1         2             1                 3
p2         4             3                 7

Average Waiting Time=1.333333
Average Turnaround Time=3.666667[root@localhost ~]#

```

4 c) Write an Priority scheduling program in C to determine the average waiting time and average turnaround time given n processes and their burst times.

Program

```

#include <stdio.h>

//Function to swap two variables
void swap(int *a,int *b)
{
    int temp=*a;
    *a=*b;
    *b=temp;
}
int main()
{
    int n;

```

```

printf("Enter Number of Processes: ");
scanf("%d",&n);

// b is array for burst time, p for priority and index for process id
int b[n],p[n],index[n];
for(int i=0;i<n;i++)
{
    printf("Enter Burst Time and Priority Value for Process %d: ",i+1);
    scanf("%d %d",&b[i],&p[i]);
    index[i]=i+1;
}
for(int i=0;i<n;i++)
{
    int a=p[i],m=i;

    //Finding out highest priority element and placing it at its desired position
    for(int j=i;j<n;j++)
    {
        if(p[j] > a)
        {
            a=p[j];
            m=j;
        }
    }

    //Swapping processes
    swap(&p[i], &p[m]);
    swap(&b[i], &b[m]);
    swap(&index[i],&index[m]);
}

// T stores the starting time of process
int t=0;

//Printing scheduled process
printf("Order of process Execution is\n");
for(int i=0;i<n;i++)
{
    printf("P%d is executed from %d to %d\n",index[i],t,t+b[i]);
    t+=b[i];
}
printf("\n");
printf("Process Id   Burst Time   Wait Time   TurnAround Time\n");
int wait_time=0;
for(int i=0;i<n;i++)
{
    printf("P%d %d%d%d\n",index[i],b[i],wait_time,wait_time + b[i]);
    wait_time += b[i];
}
return 0;

```

```
}
```

Output

Enter Number of Processes: 3
Enter Burst Time and Priority Value for Process 1: 10 2
Enter Burst Time and Priority Value for Process 2: 5 0
Enter Burst Time and Priority Value for Process 3: 8 1
Order of process Execution is
P1 is executed from 0 to 10
P3 is executed from 10 to 18
P2 is executed from 18 to 23

Process Id	Burst Time	Wait Time	TurnAround Time
P1	10	0	10
P3	8	10	18
P2	5	18	23

4 d) Write an Round Robin scheduling program in C to determine the average waiting time and average turnaround time given n processes and their burst times.

Program

```
#include<stdio.h>

int main()
{
    //Input no of processed
    int n;
    printf("Enter Total Number of Processes:");
    scanf("%d", &n);
    int wait_time = 0, ta_time = 0, arr_time[n], burst_time[n], temp_burst_time[n];
    int x = n;

    //Input details of processes
    for(int i = 0; i < n; i++)
    {
        printf("Enter Details of Process %d \n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &arr_time[i]);
        printf("Burst Time: ");
        scanf("%d", &burst_time[i]);
        temp_burst_time[i] = burst_time[i];
    }

    //Input time slot
    int time_slot;
```

```

printf("Enter Time Slot:");
scanf("%d", &time_slot);

//Total indicates total time
//counter indicates which process is executed
int total = 0, counter = 0,i;
printf("Process ID    Burst Time    Turnaround Time    Waiting Time\n");
for(total=0, i = 0; x!=0; )
{
    // define the conditions
    if(temp_burst_time[i] <= time_slot && temp_burst_time[i] > 0)
    {
        total = total + temp_burst_time[i];
        temp_burst_time[i] = 0;
        counter=1;
    }
    else if(temp_burst_time[i] > 0)
    {
        temp_burst_time[i] = temp_burst_time[i] - time_slot;
        total += time_slot;
    }
    if(temp_burst_time[i]==0 && counter==1)
    {
        x--; //decrement the process no.
        printf("\nProcess No %d \t\t %d\t\t\t %d\t\t\t %d", i+1, burst_time[i],
            total-arr_time[i], total-arr_time[i]-burst_time[i]);
        wait_time = wait_time+total-arr_time[i]-burst_time[i];
        ta_time += total -arr_time[i];
        counter =0;
    }
    if(i==n-1)
    {
        i=0;
    }
    else if(arr_time[i+1]<=total)
    {
        i++;
    }
    else
    {
        i=0;
    }
}
float average_wait_time = wait_time * 1.0 / n;
float average_turnaround_time = ta_time * 1.0 / n;
printf("\nAverage Waiting Time:%f", average_wait_time);
printf("\nAvg Turnaround Time:%f", average_turnaround_time);
return 0;
}

```

Output

Enter Total Number of Processes:3

Enter Details of Process 1

Arrival Time: 0

Burst Time: 10

Enter Details of Process 2

Arrival Time: 1

Burst Time: 8

Enter Details of Process 3

Arrival Time: 2

Burst Time: 7

Enter Time Slot:5

Process ID	Burst Time	Turnaround Time	Waiting Time
Process No 1	10	20	10
Process No 2	8	22	14
Process No 3	7	23	16

Average Waiting Time: 13.333333

Avg Turnaround Time: 21.666666

Ex No :5. Write an algorithm to synchronize the agent and the smokers using semaphore.

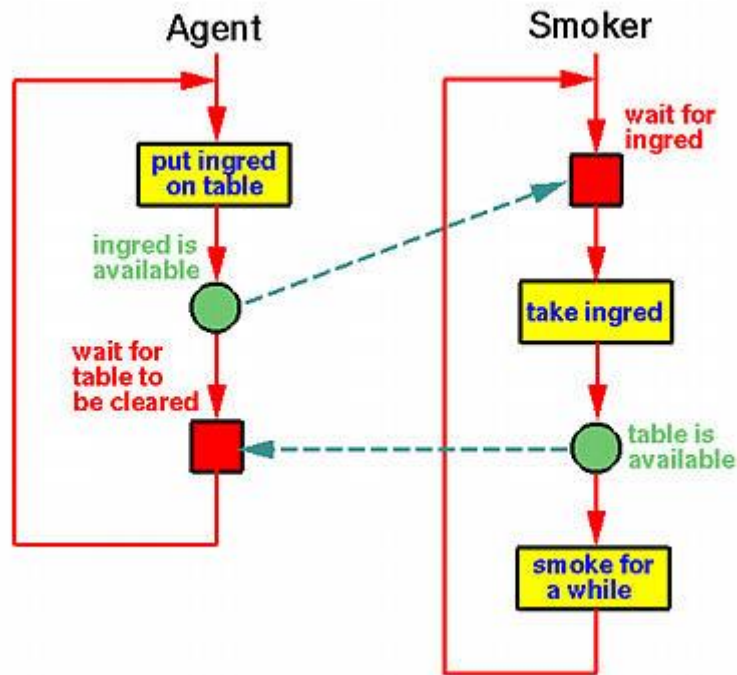
Problem

Suppose a cigarette requires three ingredients, tobacco, paper and match. There are three chain smokers. Each of them has only one ingredient with infinite supply. There is an agent who has infinite supply of all three ingredients. To make a cigarette, the smoker has tobacco (resp., paper and match) must have the other two ingredients paper and match (resp., tobacco and match, and tobacco and paper). The agent and smokers share a table. The agent randomly generates two ingredients and notifies the smoker who needs these two ingredients. Once the ingredients are taken from the table, the agent supplies another two. On the other hand, each smoker waits for the agent's notification. Once it is notified, the smoker picks up the ingredients, makes a cigarette, smokes for a while, and goes back to the table waiting for his next ingredients.

Write a program that simulates this system, with three smokers and the agent being simulated by threads.

Analysis

Let us draw a diagram that details all activities of the agent and a smoker. As usual, we use red squares and green circles for semaphore waits and signals, respectively. The table is a shared object, because the agent puts ingredients on the table and smokers take ingredients from the table. Therefore, some protection is necessary to avoid race condition. Is a mutex lock good enough? A good question. Isn't it? No, a mutex lock simply does not work, because with a lock the owner must unlock the lock which is not the case here. The agent waits for the table to become available, and then puts ingredients on the table. If we use a mutex lock, the agent must unlock the table, and if the agent is a fast runner he may come back and lock the table again to make another set of ingredient before the previous ones are taken. Thus, we have a race condition. Thus, we use a semaphore with which the thread that executes a wait to lock the semaphore and the thread that executes a signal to unlock the semaphore do not have to be the same.



When the table is available, the agent puts ingredient on the table. Of course, the agent knows what ingredients are on the table, and, hence, signals the smoker that needs this set of ingredients. On the other hand, a smoker cannot continue if he has no ingredients. To this end, we need another semaphore to block a smoker. In fact, we need three semaphores, one for each smoker. For example, if the agent puts paper and match on the table, he must inform the smoker who needs paper and match to continue. Therefore, the smoker who needs paper and match must wait on this semaphore.

The meaning of the above diagram is clear. The agent prepares for the ingredients, waits on the table, makes the ingredients available on the table when it becomes available, signals the proper smoker to take the ingredient, and then goes back for another round. For a smoker, he waits for the ingredients he needs, takes them when they are available, informs the agent that the table is free, makes a cigarette and smokes, and goes back for another round.

From the above discussion, we see that the table is not protected with a lock. Instead, it is protected through notification!

Program:

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <unistd.h>

```

```

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t smoker = PTHREAD_MUTEX_INITIALIZER;

```

```
//actors conditionals
```

```

pthread_cond_t agent_c = PTHREAD_COND_INITIALIZER;
pthread_cond_t smoker_tobacco_c = PTHREAD_COND_INITIALIZER;
pthread_cond_t smoker_match_c = PTHREAD_COND_INITIALIZER;

```

```
pthread_cond_t smoker_paper_c = PTHREAD_COND_INITIALIZER;
```

```
//resource conditionals
```

```
pthread_cond_t tobacco = PTHREAD_COND_INITIALIZER;
```

```
pthread_cond_t paper = PTHREAD_COND_INITIALIZER;
```

```
pthread_cond_t match = PTHREAD_COND_INITIALIZER;
```

```
int have_tobacco = 0;
```

```
int have_paper = 0;
```

```
int have_match = 0;
```

```
int agent_job = 1;
```

```
int smoker_tobacco_job = 0;
```

```
int smoker_match_job = 0;
```

```
int smoker_paper_job = 0;
```

```
int getRand(int range){
```

```
    int randNum = rand() % range;
```

```
    return randNum;
```

```
}
```

```
void * agent(void * arg){
```

```
    while(1) {
```

```
        sleep(1);
```

```
        pthread_mutex_lock(&m);
```

```
        //The agent stays waiting if agent_job is equal 0
```

```
        while(agent_job == 0)
```

```
            pthread_cond_wait(&agent_c, &m);
```

```
        printf("-----\n");
```

```
        int randNum = getRand(3);
```

```
        //Paper and Match
```

```
        if ( randNum == 0 ) {
```

```
            agent_job = 0;
```

```
            have_match = 1;
```

```
            have_paper = 1;
```

```
            puts("Put paper and match");
```

```
            pthread_cond_signal(&paper);
```

```
            pthread_cond_signal(&match);
```

```

    }
    //Tobacco and Match
    else if ( randNum == 1 ) {
        agent_job = 0;
        have_match = 1;
        have_tobacco = 1;
        puts("Put tobacco and match");
        pthread_cond_signal(&paper);
        pthread_cond_signal(&match);
    }
    //Tobacco and Paper
    else if ( randNum == 2 ) {
        agent_job = 0;
        have_tobacco = 1;
        have_paper = 1;
        puts("Put paper and tobacco");
        pthread_cond_signal(&paper);
        pthread_cond_signal(&tobacco);
    }

    pthread_mutex_unlock(&m);
}
return 0;
}

void * pusher_paper(void * arg){

while(1){
    pthread_mutex_lock(&m);
    while(have_paper == 0)
        pthread_cond_wait(&paper, &m);

    if(have_match == 1) {
        have_match = 0;
        agent_job = 0;
        smoker_tobacco_job = 1;
        puts("Call the tobacco smoker");
        pthread_cond_signal(&smoker_tobacco_c);
    }
    if(have_tobacco == 1) {
        have_tobacco = 0;
        agent_job = 0;
        smoker_match_job = 1;
        puts("Call the match smoker");
        pthread_cond_signal(&smoker_match_c);
    }
    pthread_mutex_unlock(&m);
}
}

```

```

    return 0 ;
}

void * pusher_match(void * arg){

    while(1) {
        pthread_mutex_lock(&m);
        while(have_match == 0)
            pthread_cond_wait(&match, &m);

        if(have_paper == 1) {
            have_paper = 0;
            agent_job = 0;
            smoker_tobacco_job = 1;
            puts("Call the tobacco smoker");
            pthread_cond_signal(&smoker_tobacco_c);
        }
        if(have_tobacco == 1) {
            have_tobacco = 0;
            agent_job = 0;
            smoker_paper_job = 1;
            puts("Call the paper smoker");
            pthread_cond_signal(&smoker_paper_c);
        }
        pthread_mutex_unlock(&m);
    }

    return 0 ;
}

void * pusher_tobacco(void * arg){
    while(1){
        pthread_mutex_lock(&m);
        while(have_tobacco == 0)
            pthread_cond_wait(&tobacco, &m);

        if(have_match == 1) {
            have_match = 0;
            agent_job = 0;
            smoker_paper_job = 1;
            puts("Call the paper smoker");
            pthread_cond_signal(&smoker_paper_c);
        }
        if(have_paper == 1) {
            have_tobacco = 0;
            agent_job = 0;
            smoker_match_job = 1;
            puts("Call the match smoker");
            pthread_cond_signal(&smoker_match_c);
        }
    }
}

```

```

    pthread_mutex_unlock(&m);
}
return 0 ;
}

void * smoker_tobacco(void * arg){

    while(1){

        pthread_mutex_lock(&smoker);
        while(smoker_tobacco_job == 0)
            pthread_cond_wait(&smoker_tobacco_c, &smoker);
        have_paper = 0;
        have_match = 0;
        smoker_tobacco_job = 0;
        agent_job = 1;
        puts("Tobacco Smoker: make cigarette...");
        pthread_mutex_unlock(&smoker);

        puts("Tobacco Smoker: Smoking...");
    }

    return 0;
}

void * smoker_paper(void * arg){

    while(1){

        pthread_mutex_lock(&smoker);
        while(smoker_paper_job == 0)
            pthread_cond_wait(&smoker_paper_c, &smoker);
        have_tobacco = 0;
        have_match = 0;
        smoker_paper_job = 0;
        agent_job = 1;
        puts("Paper Smoker: make cigarette...");
        pthread_mutex_unlock(&smoker);

        puts("Paper Smoker: Smoking...");
    }

    return 0;
}

void * smoker_match(void * arg){

    while(1){

        pthread_mutex_lock(&smoker);

```

```

        while(smoker_match_job == 0)
            pthread_cond_wait(&smoker_match_c, &smoker);
        have_paper = 0;
        have_tobacco = 0;
        smoker_match_job = 0;
        agent_job = 1;
        puts("Match Smoker: make cigarette...");
        pthread_mutex_unlock(&smoker);

        puts("Match Smoker: Smoking...");
    }

    return 0;
}

int main(int argc, char *argv[])
{
    pthread_t agent_t, smoker_tobacco_t, smoker_paper_t, smoker_match_t,
    pusher_tobacco_t, pusher_paper_t, pusher_match_t;

    //random seed
    time_t t;
    srand((unsigned) time(&t));

    if (pthread_create(&agent_t, NULL, agent, NULL) != 0) {
        fprintf(stderr, "Impossible to create thread\n");
        exit(1);
    }

    if (pthread_create(&pusher_tobacco_t, NULL, pusher_tobacco, NULL) != 0) {
        fprintf(stderr, "Impossible to create thread\n");
        exit(1);
    }

    if (pthread_create(&pusher_paper_t, NULL, pusher_paper, NULL) != 0) {
        fprintf(stderr, "Impossible to create thread\n");
        exit(1);
    }

    if (pthread_create(&pusher_match_t, NULL, pusher_match, NULL) != 0) {
        fprintf(stderr, "Impossível criar thread\n");
        exit(1);
    }

    if (pthread_create(&smoker_tobacco_t, NULL, smoker_tobacco, NULL) != 0) {
        fprintf(stderr, "Impossible to create thread\n");
        exit(1);
    }
}

```

```

if (pthread_create(&smoker_paper_t, NULL, smoker_paper, NULL) != 0) {
    fprintf(stderr, "Impossible to create thread\n");
    exit(1);
}

if (pthread_create(&smoker_match_t, NULL, smoker_match, NULL) != 0) {
    fprintf(stderr, "Impossible to create thread\n");
    exit(1);
}

pthread_join(agent_t, NULL);
pthread_join(push_tobacco_t, NULL);
pthread_join(push_paper_t, NULL);
pthread_join(push_match_t, NULL);
pthread_join(smoker_tobacco_t, NULL);
pthread_join(smoker_paper_t, NULL);
pthread_join(smoker_match_t, NULL);
}

```

Output:

☒
☒
☒

input

```

-----
Put paper and tobacco
Call the match smoker
Match Smoker: make cigarette...
Match Smoker: Smoking...
-----

Put paper and match
Call the tobacco smoker
Tobacco Smoker: make cigarette...
Tobacco Smoker: Smoking...
-----

Put paper and tobacco
Call the match smoker
Match Smoker: make cigarette...
Match Smoker: Smoking...
-----

Put paper and tobacco
Call the match smoker
Match Smoker: make cigarette...
Match Smoker: Smoking...
-----

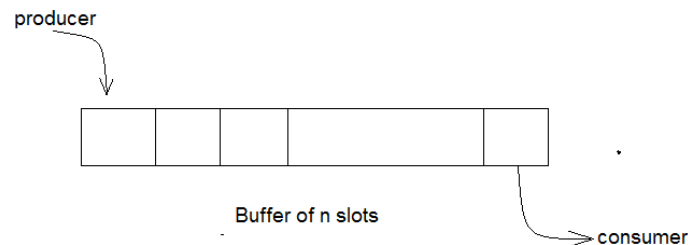
Put paper and tobacco
Call the match smoker
Match Smoker: make cigarette...
Match Smoker: Smoking...
-----

Put paper and match

```


Ex No :6. Write a C program to implement Producer–Consumer problem with Bounded Buffer

Bounded buffer problem, which is also called producer consumer problem, is one of the classic problems of synchronization.



A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- `m`, a binary semaphore which is used to acquire and release the lock.
- `empty`, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- `full`, a counting semaphore whose initial value is 0.

At any instant, the current value of `empty` represents the number of empty slots in the buffer and `full` represents the number of occupied slots in the buffer.

Program

```
#include<stdio.h>
#include<stdlib.h>

int mutex=1,full=0,empty=3,x=0;

int main()
{
    int n;
    void producer();
    void consumer();
    int wait(int);
    int signal(int);
    printf("\n1.Producer\n2.Consumer\n3.Exit");
    while(1)
    {
        printf("\nEnter your choice:");
        scanf("%d",&n);
        switch(n)
        {
```

```

case 1: if((mutex==1)&&(empty!=0))
producer();
else
printf("Buffer is full!!");
break;
case 2: if((mutex==1)&&(full!=0))
consumer();
else
printf("Buffer is empty!!");
break;
case 3:
exit(0);
break;
}
}
return 0;
}

```

```

int wait(int s)
{
return (--s);
}

```

```

int signal(int s)
{
return(++s);
}

```

```

void producer()
{
mutex=wait(mutex);
full=signal(full);
empty=wait(empty);
x++;
printf("\\nProducer produces the item %d",x);
mutex=signal(mutex);
}

```

```

void consumer()
{
mutex=wait(mutex);
full=wait(full);
empty=signal(empty);
printf("\\nConsumer consumes item %d",x);
x--;
mutex=signal(mutex);
}

```

```

}

```

Output

```
1.Producer
2.Consumer
3.Exit
Enter your choice:1
Producer produces the item 1
Enter your choice:2
Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:1
Producer produces the item 1
Enter your choice:1
Producer produces the item 2
Enter your choice:1

Producer produces the item 3
Enter your choice:1
Buffer is full!!
Enter your choice:3
```

Ex No :7. Write a c program to implement Dining–Philosopher Problem

The Dining Philosopher Problem is a classic synchronization problem that involves multiple processes (philosophers) sharing a limited set of resources (forks) in order to perform a task (eating). In order to avoid deadlock or starvation, a solution must be implemented that ensures that each philosopher can access the resources they need to perform their task without interference from other philosophers.

One common solution to the Dining Philosopher Problem uses semaphores, a synchronization mechanism that can be used to control access to shared resources. In this solution, each fork is represented by a semaphore, and a philosopher must acquire both the semaphore for the fork to their left and the semaphore for the fork to their right before they can begin eating. If a philosopher cannot acquire both semaphores, they must wait until they become available.

Algorithm

1. Initialize the semaphores for each fork to 1 (indicating that they are available).
2. Initialize a binary semaphore (mutex) to 1 to ensure that only one philosopher can attempt to pick up a fork at a time.
3. For each philosopher process, create a separate thread that executes the following code:
 - While true:
 - Think for a random amount of time.
 - Acquire the mutex semaphore to ensure that only one philosopher can attempt to pick up a fork at a time.
 - Attempt to acquire the semaphore for the fork to the left.
 - If successful, attempt to acquire the semaphore for the fork to the right.
 - If both forks are acquired successfully, eat for a random amount of time and then release both semaphores.
 - If not successful in acquiring both forks, release the semaphore for the fork to the left (if acquired) and then release the mutex semaphore and go back to thinking.
4. Run the philosopher threads concurrently.

By using semaphores to control access to the forks, the Dining Philosopher Problem can be solved in a way that avoids deadlock and starvation. The use of the mutex semaphore ensures that only one philosopher can attempt to pick up a fork at a time, while the use of the fork semaphores ensures that a philosopher can only eat if both forks are available.

Overall, the Dining Philosopher Problem solution using semaphores is a classic example of how synchronization mechanisms can be used to solve complex synchronization problems in concurrent programming.

Program

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define N 5
#define THINKING 2
#define HUNGRY 1
```

```

#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
    if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        // state that eating
        state[phnum] = EATING;

        sleep(2);

        printf("Philosopher %d takes fork %d and %d\n",
               phnum + 1, LEFT + 1, phnum + 1);

        printf("Philosopher %d is Eating\n", phnum + 1);

        // sem_post(&S[phnum]) has no effect
        // during takefork
        // used to wake up hungry philosophers
        // during putfork
        sem_post(&S[phnum]);
    }
}

// take up chopsticks
void take_fork(int phnum)
{
    sem_wait(&mutex);

    // state that hungry
    state[phnum] = HUNGRY;

    printf("Philosopher %d is Hungry\n", phnum + 1);

    // eat if neighbours are not eating
    test(phnum);

    sem_post(&mutex);

    // if unable to eat wait to be signalled

```

```

        sem_wait(&S[phnum]);

        sleep(1);
    }

// put down chopsticks
void put_fork(int phnum)
{
    sem_wait(&mutex);

    // state that thinking
    state[phnum] = THINKING;

    printf("Philosopher %d putting fork %d and %d down\n",
           phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);

    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}

void* philosopher(void* num)
{
    while (1) {
        int* i = num;
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}

int main()
{
    int i;
    pthread_t thread_id[N];

    // initialize the semaphores
    sem_init(&mutex, 0, 1);

    for (i = 0; i < N; i++)
        sem_init(&S[i], 0, 0);

    for (i = 0; i < N; i++) {
        // create philosopher processes

```

```

        pthread_create(&thread_id[i], NULL,
                      philosopher, &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }

    for (i = 0; i < N; i++)
        pthread_join(thread_id[i], NULL);
}

```

Output:

```

[root@localhost ~]# ./dining
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 5 is Hungry
Philosopher 4 is Hungry
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 4 is Hungry
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking

```

Ex No :8. Write an algorithm for synchronization between reader processes and write processes using semaphore.

The readers-writers problem is a classical problem of process synchronization, it relates to a data set such as a file that is shared between more than one process at a time. Among these various processes, some are Readers - which can only read the data set; they do not perform any updates, some are Writers - can both read and write in the data sets. The readers-writers problem is used for managing synchronization among various reader and writer process so that there are no problems with the data sets, i.e. no inconsistency is generated.

Let's understand with an example - If two or more than two readers want to access the file at the same point in time there will be no problem. However, in other situations like when two writers or one reader and one writer wants to access the file at the same point of time, there may occur some problems, hence the task is to design the code in such a manner that if one reader is reading then no writer is allowed to update at the same point of time, similarly, if one writer is writing no reader is allowed to read the file at that point of time and if one writer is updating a file other writers should not be allowed to update the file at the same point of time. However, multiple readers can access the object at the same time.

Let us understand the possibility of reading and writing with the table given below:

Case	Process 1	Process 2	Allowed / Not Allowed
Case 1	Writing	Writing	Not Allowed
Case 2	Reading	Writing	Not Allowed
Case 3	Writing	Reading	Not Allowed
Case 4	Reading	Reading	Allowed

Algorithm:

1. Initialize semaphores for mutual exclusion (`mutex`) and control of reading/writing (`rw_mutex`, `wrt`).
2. Create reader and writer processes.
3. For the writer process:
 - Wait for the `wrt` semaphore to allow writing (blocks readers from accessing the shared resource).
 - Perform the writing operation.
 - Signal the `wrt` semaphore to allow readers to access the shared resource.
4. For the reader process:
 - Wait for the `mutex` semaphore to ensure exclusive access to the reader count.
 - Increment the reader count.

- If it's the first reader, wait on the `wrt` semaphore to block writers.
- Signal the `mutex` semaphore to allow other readers to update the count.
- Read from the shared resource.
- Wait on `mutex` semaphore to update the reader count.
- Decrement the reader count.
- If it's the last reader, signal the `wrt` semaphore to allow writers.
- Signal the `mutex` semaphore to allow other processes to access the reader count.

Program:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex, rw_mutex, wrt;
int reader_count = 0;
int data = 0;

void *writer(void *arg) {
    while (1) {
        // Writer enters critical section
        sem_wait(&wrt);
        data++; // Writing operation
        printf("Data written by writer: %d\n", data);
        sem_post(&wrt); // Writer leaves critical section
        sleep(1); // Simulating delay between write operations
    }
}

void *reader(void *arg) {
    while (1) {
        sem_wait(&mutex); // Ensures exclusive access to reader count
        reader_count++;
        if (reader_count == 1) {
            sem_wait(&wrt); // Blocks writers if it's the first reader
        }
    }
}
```

```

sem_post(&mutex); // Release mutex for reader count

// Reading operation
printf("Data read by reader: %d\n", data);
sleep(1); // Simulating delay for reading

sem_wait(&mutex); // Update reader count
reader_count--;
if (reader_count == 0) {
    sem_post(&wrt); // Signal writer if it's the last reader
}
sem_post(&mutex); // Release mutex for reader count
}
}

int main() {
    // Initializing semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&rw_mutex, 0, 1);
    sem_init(&wrt, 0, 1);

    // Creating threads for reader and writer
    pthread_t writerThread, readerThread;
    pthread_create(&writerThread, NULL, writer, NULL);
    pthread_create(&readerThread, NULL, reader, NULL);

    // Joining threads
    pthread_join(writerThread, NULL);
    pthread_join(readerThread, NULL);

    // Destroying semaphores
    sem_destroy(&mutex);
    sem_destroy(&rw_mutex);
    sem_destroy(&wrt);

    return 0;
}

```

Output:

```
[root@localhost ~]# gcc -o ex8 ex8.c -lpthread
[root@localhost ~]# ./ex8
Data read by reader: 0
Data read by reader: 0
Data written by writer: 1
Data read by reader: 1
Data written by writer: 2
Data read by reader: 2
Data written by writer: 3
Data read by reader: 3
Data read by reader: 3
Data written by writer: 4
Data read by reader: 4
Data written by writer: 5
Data read by reader: 5
Data read by reader: 5
Data written by writer: 6
Data read by reader: 6
Data read by reader: 6
```